Frame: An Imperative Coordination Language for Parallel Programming

Division of Informatics, University of Edinburgh

Institute for Computing Systems Architecture

Frame: An Imperative Coordination Language for Parallel Programming

by

Murray Cole

Informatics Research Report EDI-INF-RR-0026

Division of Informatics **September 2000**
http://www.informatics.ed.ac.uk/

Frame: An Imperative Coordination Language for Parallel Programming

Murray Cole

Informatics Research Report EDI-INF-RR-0026

DIVISION *of* INFORMATICS
Institute for Computing Systems Architecture

September 2000

Abstract :

We present Frame, a simple language which facilitates structured expression of imperative parallelism. Programs are described at two levels. The top level captures the main parallel algorithmic structure (which may be nested) and is independent of the language used in the lower level to describe the building blocks of sequential or unstructured parallel code which it coordinates. In the current instantiation of Frame the lower level code is expressed in C with calls to MPI. Frame exists as a simple demonstration of the principle of combining nested imperative parallel control structure with properly contained ad-hoc parallelism. Subsequent languages in the Frame family should augment it with further control constructs, more sophisticated interfaces to the ad-hoc level and optimised implementation strategies.

Keywords : skeletal parallel programming

Frame: An Imperative Coordination Language for Parallel Programming

Murray Cole
Institute for Computing Systems Architecture
Division of Informatics
University of Edinburgh

Abstract

We present Frame, a simple language which facilitates structured expression of imperative parallelism. Programs are described at two levels. The top level captures the main parallel algorithmic structure (which may be nested) and is independent of the language used in the lower level to describe the building blocks of sequential or unstructured parallel code which it coordinates. In the current instantiation of Frame the lower level code is expressed in C with calls to MPI. Frame exists as a simple demonstration of the principle of combining nested imperative parallel control structure with properly contained *ad-hoc* parallelism. Subsequent languages in the Frame family should augment it with further control constructs, more sophisticated interfaces to the ad-hoc level and optimised implementation strategies.

1 Introduction

The purpose of this paper is to argue the case for a new class of parallel programming language. We believe that a careful combination of the pragmatism of currently popular languages such as MPI[15], OpenMP[16] and thread based systems with the principles of the skeletal programming approach can produce a framework which benefits from the the popular appeal of the former and the conceptual simplicity of the latter. We begin with a brief review of existing approaches. We then describe Frame, a very simple instance of the class of languages we envisage. We stress from the outset that Frame is merely intended as a concrete proof of concept. Much more work is required to develop the successor languages which may hope to gain the acceptance of the parallel programming community.

2 Current Frameworks

2.1 Micro-management of Parallelism

Many prevalent systems for parallel programming, whether based around message passing or shared memory, have a common weakness: the overall conceptual algorithmic structure cannot be discerned by inspection of the source code. It exists as a coherent whole only in the programmer's head, and becomes embedded in source code implicitly, within the myriad interactions between calls to communication primitives, thread operations, semaphores, locks and so on. This micro-management of parallel structure causes several problems:

- portability is inhibited, since it is impossible to systematically transfer the essential algorithmic structure of a program to a new software environment (for example, from MPI to Java threads) because that structure is impossible to disentangle from the primitives of the old environment;
- compiler optimisations are inhibited, since the programmer's expert understanding of the program's overall structure and interactions between its components are lost (in fact they are never explictly stated) and are impossible, in general, to reconstruct by inspection of the source code;
- development time is increased, since the programmer has to wrestle with the interactions between primitives which collectively express the big picture. Changing the overall algorithmic structure is error-prone.

Such issues are partially addressed by mechanisms such as the collective communication operations of MPI, but these only assist individual program steps.

2.2 Annotations and Directives

The program annotations of OpenMP are, from our perspective, a positive step. The programmer makes concise statements concerning the intended parallel structure which the compiler has the task of realising in detail. The use of sequential code in familiar languages as the target of the annotations is clearly appealing. On the other hand,

- the operations and semantics are deliberately tied to a shared memory execution model, reducing the ease with which algorithmic structure could be ported more widely (for example to MPI);
- the syntax and semantics of nested directives are complex and poorly defined, encouraging a programming style in which conceptual structure is flattened and made implicit in lower level interactions;
- the constructs are rather limited, consisting of `for` loop distribution and arbitrary concurrent sections together with an ad-hoc, fixed selection of reductions.

2.3 Skeletal Languages

Skeletal programming models [3, 5, 9, 14, 17] are by definition strong in terms of facilities for expressing high-level structure. The natural language paradigm within which to express such structure is functional. However, it is interesting to note that that the skeletal projects which have arguably made most ground on a wider stage [1, 4, 9] either use languages which are imperative in syntactic flavour, if not necessarily semantics, or encourage integration of fragments of sequential imperative code, or both. A valid criticism of the skeletal approach is that some parallel algorithms are relatively unstructured, either in coarse or fine detail and that a methodology which prohibits expression of such computations is inherently restricted in application.

3 Structured Imperative Coordination Languages

We now outline the characteristics of a new branch of the skeletal family tree. Our aim is to provoke research into a class of languages which may further popularise the principles of skeletal parallelism. To attract the interst of the imperative parallel programmer, we speculate that we must embed skeletal ideas in familiar syntactic and semantic frameworks and that we must offer the option of dropping into the unstructured world beneath. Thus we argue here for new instantiations of the skeletal approach, characterised by

- parallel control constructs dealing purely with patterns of control (rather than data), fullly nestable in block structured style;
- a facility for invoking operations from an underlying parallel software framework (for example, MPI) within carefully contained conceptual boundaries;
- a semantics for the control constructs which is orthogonal to the semantics of the lower level function calls.

By way of example, we describe a very simple language, Frame which provides just such a facility, building on C/MPI as the supporting ad-hoc parallel layer. We emphasise that the high level sections of our programs are independent of this choice and would be directly portable to implementations on other supporting layers. Our proposal is related to work by Crooke [8] and Marr [13] who investigated the integration of skeletal frameworks with imperative base languages and in the latter case allowed the introduction of controlled ad-hoc parallelism. However, both systems characterised skeletons as functions applied to code and data and returning data. In contrast, the skeletons of Frame are straightforward control constructs in the traditional imperative sense.

4 What about Object Orientation?

Conventional software engineering continues to progress and structured programming and languages are old hat. However, while current methodologies focus on issues of

programming in-the-large, the components they organise continue to be coded in-the-small with familiar constructs (consider Java's syntactic relationship to C). There is a proper place for development of in-the-large methodologies for parallelism (probably they are the same as for the sequential case). Here, we offer some thoughts on tackling the question of finding a correctly structured in-the-small framework.

5 Frame Concepts

A Frame program has two sections. The first describes the top-down parallel structure of the intended computation. The second is a collection of function definitions, written in the 'base" parallel language, which can be called from the first section. The base language is a standard parallel programming language, augmented with a small number of reserved identifiers (capturing concepts of process identifier and group membership) and restricted to enforce the principle of well-contained ad-hoc parallelism. In a later section, we will illustrate the approach with C/MPI as the base language.

All that we require of a base model is that it have a well defined concept of process. A fixed (power of two) number of processes are created at the start of a run and remain in existence until program termination. There is no dynamic process creation (at least none that is visible to the upper layer). Frame programs inherit their declaration and interaction mechanisms from the base language. The upper layer adds a facility for structuring the patterns and groupings within which these interactions take place. To the programmer, they are shorthand forms for the patterns of interaction which would otherwise have to be achieved through disciplined use of the base level primitives.

The upper layer of Frame is block-structured and imperative. Statements sequences contain instances of the three Frame control constructs and calls to base level functions. Operationally, all processes begin in a single group, executing the first statement in the program. There is barrier synchronisation between statements. The `ind` statement indicates that the associated block should be executed by processes in independent groups of size one. The `con` (for "conquer") construct dictates a pattern in which processes are first grouped in pairs, then fours, eights and so on until all processes are involved in a final group. In contrast, the `div` (for "divide") construct reverses this pattern, starting with a group in which all processes are active, then two concurrent groups of half the size, then four of a quarter the size and so on, until a final step with groups of size two. The constructs are freely nestable and operate with respect to the group of processors currently in context. Similarly, base level functions are called with respect to the current grouping. To the called function, the current group appears to be the whole machine.

6 Frame Language Definition

A complete program includes text in both the base level language and the new upper level constructs provided by Frame itself, with a syntactically enforced separation

between the two. The interface between these layers is discussed in section 7.

6.1 Syntax

program	::=	*statement* ˜ *lowerlayer*	
statement	::=	*statement* *statement*	|
		<*basecall*>	|
		ind { *statement* }	|
		con { *statement* }	|
		div { *statement* }	|

Figure 1: Frame syntax

The syntax of Frame is presented in figure 1. The *lowerlayer* consists of a collection of function and data declarations in the base language. Similarly, *basecall* indicates a call to one of the base language functions

6.2 Semantics

Just as Frame coordinates activities expressed in the base language, so a semantics for the upper layer must be defined on top of the semantics of the base layer. We begin by stating the properties we require of a suitable base semantics. We have already noted that program execution will activate a fixed number of processes, p. We assume that these are permanently indexed from 0 to $p-1$. A *configuration* of these processes is an indication of the way they are grouped together, and of their activity or inactivity. Semantically, configurations have type C

$$C = Nat \rightarrow \{Nat\}$$

indicating for each process, the group of processes to which it belongs. An empty group indicates inactivity (including the possibility that the index does not correspond to an actual process). At the base level we assume the existence of a semantic function $[\![\ ]\!]_{BS}$ determining the meaning of base level statements relative to a given configuration

$$[\![\ ]\!]_{BS} : statement \rightarrow C \rightarrow (machine \rightarrow machine)$$

where *machine* encompasses all relevant aspects of state (which are entirely the concern of the base level and its semantics). The definition of our upper layer semantic function $[\![\ ]\!]$ is presented in figure 2. We use $\cdot$ to denote functional composition. The focus of the semantics is on defining the way in which the constructs control

configurations. We introduce auxiliary definitions to simplify matters. In future extensions we may want to specify adjustments to those parts of a configuration which are not already inactive (for example, to allow conditional activity at the top level). To simplify this, we introduce the shorthand notation

$$\lambda^C i.e \equiv \lambda i.\ \textbf{if}\ C\,i\ =\ \{\}\ \textbf{then}\ \{\}\ \textbf{else}\ e$$

The fundamental operation on configurations will be to take each group and decompose it into two groups of half the size.

$$\begin{array}{lll} split\ C = \lambda^C i. & \textbf{let} & sz = |C\,i| \\ & & low = min(C\,i) \\ & & mid = low + sz/2 \\ & & high = max(C\,i) \\ & \textbf{in} & \textbf{if}\ \ sz = 2\ \ \textbf{then}\ \{\} \\ & \textbf{else} & \textbf{if}\ \ i < mid\ \ \textbf{then}\ \{low..mid-1\} \\ & \textbf{else} & \{mid..high\} \end{array}$$

The restriction on machine sizes to powers of two and the semantic behaviour of the constructs ensure that groups always consist of consecutively indexed sequences of processes and consequently that this function is always well defined.

$$\begin{array}{lcl} [\![\ s1\ s2\ ]\!]\ C & = & ([\![\ s2\ ]\!]\ C\ )\ \cdot\ ([\![\ s1\ ]\!]\ C) \\ [\![\ <\ basecall\ >\ ]\!] & = & [\![\ basecall\ ]\!]_{BS} \\ [\![\ \textbf{ind}\ \{\ s\ \}\ ]\!]\ C & = & [\![\ s\ ]\!]\ (\lambda^C i.\ \{i\}) \\ [\![\ \textbf{con}\ \{\ s\ \}\ ]\!]\ C & = & [\![\ s\ ]\!]\ C\ \cdot\ ([\![\ \textbf{con}\ \{\ s\ \}\ ]\!]\ (split\ C)) \\ [\![\ \textbf{div}\ \{\ s\ \}\ ]\!]\ C & = & ([\![\ \textbf{div}\ \{\ s\ \}\ ]\!]\ (split\ C))\ \cdot\ ([\![\ s\ ]\!]\ C) \end{array}$$

Figure 2: Frame semantics

7 Interface to MPI

The interface between Frame and MPI is simple. MPI has no concept of global data structures, and so data declarations at the base level cause the creation of variables in all processes in the usual way. In order to enforce our principle of constraining ad-hoc parallelism, we require calls to MPI communications functions to use the pre-defined communicator `mycomm`. The implementation will ensure that this encompasses precisely the current group and maintains the reserved identifiers `mygrpsize` and `myid` to provide its size and a processes index within it, dynamically. An MPI status variable `mystat` is provided similarly.

8 An Example: Bitonic Mergesort

We demonstrate our approach by expressing the well-known bitonic mergesort sorting algorithm. We begin by presenting the simple element-per-processor algorithm [2], before considering the more realistic block-based (many items per processor) variant [12]. The algorithm is a mergesort (hence the outer **con**) in which the mergestep involves some data re-organisation followed by a division process in which smaller and smaller sub-sequences are separated out (hence the inner **div**). This structure can be expressed in Frame following the analysis made in [6]. The top level of the element-per-processor program is presented in figure 3, with full code in the appendix. Notice that the semi-colons and parentheses belong to the base language syntax.

```
<load_data();>
con {
  <rev_half();>
  div {
    <bms_step ();>
  }
}
<store_data();>
```

Figure 3: Bitonic Mergesort Top Level Program

The dynamically evolving group structure created by **con** and **div** simplifies the expression of the key algorithmic steps. The `rev_half` operation reverses the order of items stored in the upper half of the group, while `bms_step ();` is the fundamental bitonic compare-exchange. In a conventional flat description (e.g. [12]) the pattern of process interaction is captured by complex and unenlightening bit operations on processor identifiers.

Switching to the block based algorithm, presented in figure 4, is simple since the algorithmic structure is identical. We only have to add an independent initial phase, in which blocks are sorted internally with sequential `quick_sort`, and to amend the reversing and compare-exchanging steps to work with blocks.

9 Rudimentary Implementations

To complete our proof-of-concept we have built a simple C/MPI implementation of Frame. The key element involves generating code to maintain a dynamically evolving stack of communication contexts, each consisting of an MPI communicator handle and the current values for the reserved identifiers. We have successfully programmed and run a number of small examples. There is considerable scope for optimisation. In particular, we note that for a given number of processors and for the simple set of constructs in the version of the language described here, the entire evolution of group structure can be determined statically. Together with analysis and amendment of the

```
<load_data();>
ind {
  <quick_sort();>
}
con {
  <rev_half();>
  div {
    <bms_step ();>
  }
}
<store_data();>
```

Figure 4: Block-based Top Level

ad-hoc parallel code, this could be exploited to remove some or all of the dynamic stacking and unstacking of communication contexts. In the extreme, we would hope to be able to statically generate a communication structure identical to that which would be expressed explicitly in a conventional approach (in other words, that there would be no overhead at all to pay for our added abstraction). Activity graphs, a candidate language independent notation for representing and manipulating such structures are considered in [7]. Student projects have investigated possible new constructs [10, 11] and an alternative implementation targeting Fork95 [11].

10 Acknowledgements

I offer my thanks to Andrea Zavanella for his contributions to our work on activity graphs and to Ruth Durie and David Fletcher for their project work.

References

[1] Bacci B. and Gorlatch S. and Lengauer C. and Pelagatti S., *Skeletons and Transformations in an Integrated Parallel Programming Environment*, in Parallel Computing Technologies (PaCT-99), Malyshkin V, (ed), Lecture Notes in Computer Science 1662, pages 13–27, Springer-Verlag, 1999.

[2] Batcher, K. *Sorting Networks and their Applications*, in Proc. AFIPS Spring Joint Computer Conference, volume 32, pages 307-314, 1968.

[3] Botorog, G. and H. Kuchen, *Efficient Parallel Programming with Algorithmic Skeletons*, in Proceedings of EuroPar '96, Lecture Notes in Computer Science vol 1123, pages 718–731, 1996.

[4] Bruce R. and Chapple S. and MacDonald N. and Trew A.S. and Trewin S., *CHIMP and PUL: Support for Portable Parallel Computing*, Fourth Annual Conference of the Meiko User Society, Southampton, 1993.

[5] Cole, M.I., *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, 1989.

[6] Cole, M.: 1997, *On Dividing and Conquering Independently*, in Proceedings of Euro-Par '97, Lecture Notes in Computer Science vol 1300, pages 634–637, 1997.

[7] Cole, M. and Zavanella, A., *Activity Graphs: A Model Independent Intermediate Layer for Skeletal Coordination*, in Proceedings of the 2000 ACM Symposium on Applied Computing, pages 255–261, 2000.

[8] Crooke D.C., *Practical Structured Parallelism Using BMF*, PhD thesis, Department of Computer Science, University of Edinburgh, 1998.

[9] Darlington, J. and Guo, Y. and To, H.W. and Yang, J., *Parallel Skeletons for Structured Composition*, in Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 19-28, ACM Press, 1995.

[10] Durie R., *Compilation of Nested Parallelism*, Honours Project Report, Division of Informatics, University of Edinburgh, 1999.

[11] Fletcher D., *Compilation of Nested Parallelism* , MSc Project Report, Division of Informatics, University of Edinburgh, 1999.

[12] Kumar, V. and Grama, A. and Gupta, A. and Karypis G. *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, 1994.

[13] Marr M.I., *Descriptive Simplicity in Parallel Computing*, PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[14] Michaelson G. and Scaife N. and Bristow P. and King P., *Nested Algorithmic Skeletons from Higher Order Functions*, to appear in Parallel Algorithms and Applications, special issue on High Level Models and Languages for Parallel Processing, 2000.

[15] Message Passing Interface Forum Website, `http://www.mpi-forum.org/`.

[16] OpenMP Architecture Review Board Website, `http://www.openmp.org/`.

[17] Pelagatti, S. and M. Danelutto: 1997, *Structured Development of Parallel Programs*. Taylor & Francis.

Appendix - Simple Bitonic Mergesort

```
<load_data();>
con {
  <rev_half();>
  div {
    <bms_step ();>
  }
}
<store_data();>

~
int *a, i, temp;

void rev_half (void)
{
  if (myid >= mygrpsize/2) {
    MPI_Sendrecv_replace(&i, 1, MPI_INT, mygrpsize-myid+(mygrpsize/2)-1,
                         0, mygrpsize-myid+(mygrpsize/2)-1, 0, mycomm, &mystat);
  }
}

void bms_step(void)
{
    temp = i;
    MPI_Sendrecv_replace(&temp, 1, MPI_INT, (myid+(mygrpsize/2))%mygrpsize,
                         0, (myid+(mygrpsize/2))%mygrpsize, 0, mycomm, &mystat);
    if ((myid < mygrpsize/2)   &&  (i > temp)) i = temp;
    if ((myid >= mygrpsize/2)  && (i < temp)) i = temp;
}

void load_data (void)
{
  FILE *f;
  int j;

  if (myid == 0) {
    a = (int *) malloc(myid*sizeof(int));
    f = fopen("/home/mic/Frame/examples/bmsinput","r");
    for (j=0; j<mygrpsize; j++) {
      fscanf(f, "%d", &a[j]);
    }
  }
  MPI_Scatter(&a[0], 1, MPI_INT, &i, 1, MPI_INT, 0, mycomm);
}
```

```
void store_data (void)
{
  int j;

  MPI_Gather(&i, 1, MPI_INT, &a[0], 1, MPI_INT, 0, mycomm);
  if (myid == 0) {
        for (j=0; j<mygrpsize; j++) {
          printf("%d ", a[j]);
        }
  }

}
```